ECE 250 Project 4
Design Document

Anthony Xu, UW User ID:a68xu

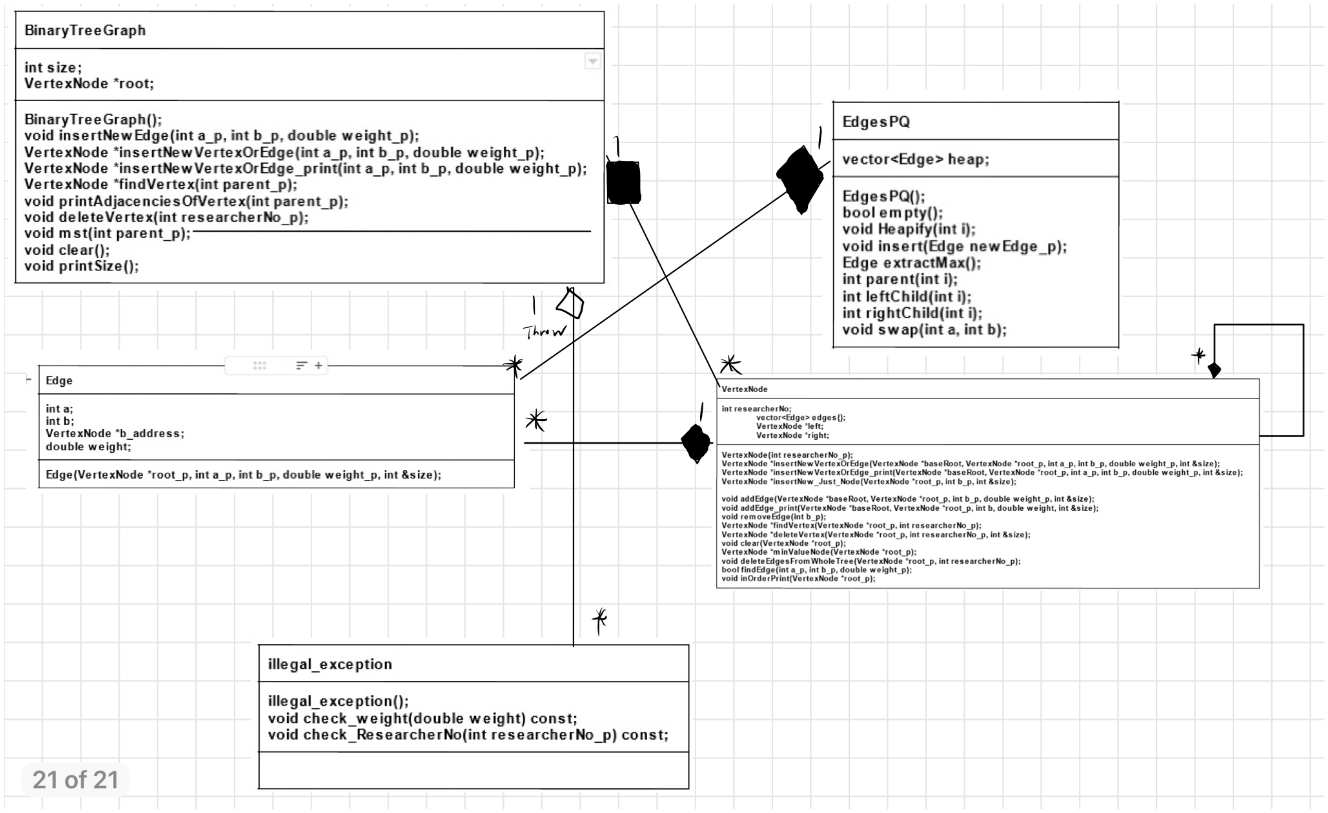
Dec 06 2022

Overview of classes:

Classes used:

- Edge
    - Contains the researcher, the researcher being influenced, and the address of the researcher being influenced as well as the weight
- VertexNode
    - An individual node that stores the researcher id, a vector set of edges, and pointers to left and right subtree
    - Supports numerous operations, mainly recursive operations
- BinaryTreeGraph
    - Contains the root node to the tree graph
    - Supports recursive operations which are called on the root node,
- Edges PQ
    - A priority queue realized by using heaps, which allows for easy extraction of finding the max weight edge when performing the MST operation on the graph
- illegal_exception
    - Allows for throwing an exception.

UML Class diagram



```
BinaryTreeGraph

int size;
VertexNode *root;

BinaryTreeGraph();
void insertNewEdge(int a_p, int b_p, double weight_p);
VertexNode *insertNewVertexOrEdge(int a_p, int b_p, double weight_p);
VertexNode *insertNewVertexOrEdge_print(int a_p, int b_p, double weight_p);
VertexNode *findVertex(int parent_p);
void printAdjacenciesOfVertex(int parent_p);
void deleteVertex(int researcherNo_p);
void mst(int parent_p);
void clear();
void printSize();
```

```
EdgesPQ

vector<Edge> heap;

EdgesPQ();
bool empty();
void Heapify(int i);
void insert(Edge newEdge_p);
Edge extractMax();
int parent(int i);
int leftChild(int i);
int rightChild(int i);
void swap(int a, int b);
```

```
Edge

int a;
int b;
VertexNode *b_address;
double weight;

Edge(VertexNode *root_p, int a_p, int b_p, double weight_p, int &size);
```

```
VertexNode

int researcherNo;
        vector<Edge> edges{};
        VertexNode *left;
        VertexNode *right;

VertexNode(int researcherNo_p);
VertexNode *insertNewVertexOrEdge(VertexNode *baseRoot, VertexNode *root_p, int a_p, int b_p, double weight_p, int &size);
VertexNode *insertNewVertexOrEdge_print(VertexNode *baseRoot, VertexNode *root_p, int a_p, int b_p, double weight_p, int &size);
VertexNode *insertNew_Just_Node(VertexNode *root_p, int b_p, int &size);

void addEdge(VertexNode *baseRoot, VertexNode *root_p, int b_p, double weight_p, int &size);
void addEdge_print(VertexNode *baseRoot, VertexNode *root_p, int b, double weight, int &size);
void removeEdge(int b_p);
VertexNode *findVertex(VertexNode *root_p, int researcherNo_p);
VertexNode *deleteVertex(VertexNode *root_p, int researcherNo_p, int &size);
void clear(VertexNode *root_p);
VertexNode *minValueNode(VertexNode *root_p);
void deleteEdgesFromWholeTree(VertexNode *root_p, int researcherNo_p);
bool findEdge(int a_p, int b_p, double weight_p);
void inOrderPrint(VertexNode *root_p);
```

```
illegal_exception

illegal_exception();
void check_weight(double weight) const;
void check_ResearcherNo(int researcherNo_p) const;
```

From: http://www.cs.utsa.edu/~cs3443/uml/uml.html

Details on design decisions

```
class Edge
{
public:
    int a;
    int b;
    VertexNode *b_address;
    double weight;
    Edge(VertexNode *root_p, int a_p, int b_p, double weight_p, int &size);
private:
};
```

Member Variables

- Id of the Researcher and the researcher being influenced
- B_address is the address of the node of the researcher being influenced, which allows for faster speed when deleting the researcher
- Weight of the edge

Member Functions

- The constructor initializes all the member variables, and then make a call to the root of the tree, to insert a new node(which is the researcher being influenced), this uses binary tree traversal to find and locate the insertion spot and see if insertion is needed

```
class VertexNode
{
    // influenced by:17569      researcher: 2629   influence weight:0.045455
public:
    int researcherNo;
    vector<Edge> edges{};
    VertexNode *left;
    VertexNode *right;

    VertexNode(int researcherNo_p);
    VertexNode *insertNewVertexOrEdge(VertexNode *baseRoot, VertexNode *root_p, int a_p, int b_p, double weight_p, int &size);
    VertexNode *insertNewVertexOrEdge_print(VertexNode *baseRoot, VertexNode *root_p, int a_p, int b_p, double weight_p, int &size);
    VertexNode *insertNew_Just_Node(VertexNode *root_p, int b_p, int &size);

    void addEdge(VertexNode *baseRoot, VertexNode *root_p, int b_p, double weight_p, int &size);
    void addEdge_print(VertexNode *baseRoot, VertexNode *root_p, int b, double weight, int &size);
    void removeEdge(int b_p);
    VertexNode *findVertex(VertexNode *root_p, int researcherNo_p);
    VertexNode *deleteVertex(VertexNode *root_p, int researcherNo_p, int &size);
    void clear(VertexNode *root_p);
    VertexNode *minValueNode(VertexNode *root_p);
    void deleteEdgesFromWholeTree(VertexNode *root_p, int researcherNo_p);
    bool findEdge(int a_p, int b_p, double weight_p);
    void inOrderPrint(VertexNode *root_p);

private:
};
```

Member variables:

- Id of the researcher
- A vector set of individual edges
- Pointers to left and right subtree of the node

Member Functions:

- insertNewVertexOrEdge first uses binary tree traversal to see if researcher a is in the tree, if not, then insert the researcher. It then checks if it is required to insert the edge by traversing the whole vector and see if the edge is already in the edge vector set. If not, then insert the edge. Then make another call to insert just the b node by calling `insertNew_Just_Node`.Traverse the tree in binary traversal, and then check if the node is already in the tree, then insert the node if the b node is not there
- Insertnewjustnode only inserts a node into the tree
- Add edge is called on an individual node and see if the edge is in the edge vector set of the node
- Removeedge removes a specific edge from the edge set of a node
- findVertex returns the address of the node according to the research id
- deleteVertex deletes a specific vertex from the tree
- Clear deletes all vertices from the tree
- minValueNode returns the address of the node that has the minimum value in a tree given a root node, allows for heap operations
- deleteEdgesFromWholeTree allows the deletion of all edges that points to a specific researchers in the whole tree
- findEdge linearly traverses the vector set of a node and tells us if a specific edge is in the vector
- inOrderPrint is for testing and allows for the sorted printing of the whole tree with the a researcher id as the key

```
class BinaryTreeGraph
{
public:
    int size;
    VertexNode *root;

    BinaryTreeGraph();
    void insertNewEdge(int a_p, int b_p, double weight_p);
    VertexNode *insertNewVertexOrEdge(int a_p, int b_p, double weight_p);
    VertexNode *insertNewVertexOrEdge_print(int a_p, int b_p, double weight_p);
    VertexNode *findVertex(int parent_p);
    void printAdjacenciesOfVertex(int parent_p);
    void deleteVertex(int researcherNo_p);
    void mst(int parent_p);
    void clear();
    void printSize();

private:
};
```

Member variables:

- The size of the tree, kept track when inserting and deleting nodes
- The pointer to the root node of the tree

Member Functions:

```
- void insertNewEdge(int a_p, int b_p, double weight_p);
- VertexNode *insertNewVertexOrEdge(int a_p, int b_p, double weight_p);
- VertexNode *insertNewVertexOrEdge_print(int a_p, int b_p, double weight_p);
- VertexNode *findVertex(int parent_p);
  void deleteVertex(int researcherNo_p);
```

Those 5 functions calls the same name function on the root node(the functions have an equal name in the VertexNode class)

- printAdjacenciesOfVertex allows for the printing of all adjacent Vertices by traversing through the vector class from beginning to end. Which corresponds to the order of insertion, since the push_back operation is called on when inserting an edge each time
- mst() uses the Prims algorithm to add up the number of nodes in the MST, which is achieved by using the priority queue class.
  We first insert all edges from the starting Vertex into the priority queue(ordered by weight),

Loop: until the priority queue is empty{
and then we use the priority queue to allow for extract the maximum weight, after extracting it, we use the address stored in the edge to insert another set of edges into the priority queue
}

```
class EdgesPQ
{
public:
    vector<Edge> heap;
    EdgesPQ();
    bool empty();
    void Heapify(int i);
    void insert(Edge newEdge_p);
    Edge extractMax();
    int parent(int i);
    int leftChild(int i);
    int rightChild(int i);
    void swap(int a, int b);

private:
};

class illegal_exception
{
public:
    illegal_exception();
    void check_weight(double weight) const;
    void check_ResearcherNo(int researcherNo_p) const;
};
```

Member variables:

- The heap uses a vector set to act as a array
- All the functions are derived from the pseudo code giving about priority queues and heaps from the lecture notes.
- Heapify builds the heap
- Extract Max extracts the maximum key(edge) and then heapify the heap again
- Additional helper functions such as , parent, leftchild, rightchild and swap to allow for cleaner code

Member Functions:

```
class illegal_exception
{
public:
    illegal_exception();
    void check_weight(double weight) const;
    void check_ResearcherNo(int researcherNo_p) const;
};
```

To allow for throwing an exception of this type

Test Cases

1. Insert with different kinds of patterns, a b w,
   a is already in the graph, b is not
   b is already in the graph, a is not
   a and b is already in the graph, w is different from previous edges
   A b w all in the graph already
   Delete vertices and try again

Run Time

V is how many vertices there are
E is how many edges there are

- Load
  - O(E) all edges we need to load into the graph
- i
  - O(logV)for average case, we would need to make height of the tree traversals to reach the spot and decided whether we need to insert Vertex a, then we check if we need to insert Vertex B(takesO(1) time on average to traverse the edge set of a node and see if we need to insert B) , if we need to insert B, then we start from the root node again which also takes the height of the tree
  - O(V*E) worst case since we are not using an AVL tree, we need to consider the worst case of the BST being like a linked list, and most of the Edges being concentrated on the node that we need to check for
- p
  - O(degree(a)) we need to first find the node which takes O(V) on average, and then we need to traverse the whole edge set of the node
  - O(V*E) worst case
- d
  - O(logN + E) we need to find the vertex that we need to delete through BST traversal and then traverse all edges in the whole graph, delete them
- mst
  - O(|E|log(|V|)) we first find the starting vertex which takes logV since we are using BST, we then insert all edges(each with a vertex) into a priority queue, we then extract the max from the priority queue and then heapify the priority queue(which takes logV). In total, we are heapifying the priority queue E times. So in total it takes ElogV to perform mst.
- Size
  - O(1) takes constant time, since we keep track of the size for all oeprations
- Exit
  - O(V) since we need to visit all Vertices in the tree once and delete them recursively, and then exit the program